CSE 521: Design and Analysis of Algorithms
Assignment #2
April 12, 2002
Due: Friday, April 19

Reading Assignment: Kleinberg and Tardos, Sections 3.1-3.2, Chapter 5

Problems:

1. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges — what's being bought — are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them — e.g. they may want to know if the four events

   `buy Yahoo, buy eBay, buy Yahoo, buy Oracle`

   occur in this sequence S, in order but not necessarily consecutively.

   They begin with a finite collection of possible *events* (e.g. the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g. `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So for example, the sequence of four events above is a subsequence of the sequence

   `buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle`

   Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time $O(m+n)$ whether S' is a subsequence of S.

2. Consider the following scheduling problem. You have a n jobs, labeled $1, \ldots, n$, which must be run one at a time, on a single processor. Job j takes time t_j to be processed. We will assume that no two jobs have the same processing time; that is, there are no two distinct jobs i and j for which $t_i = t_j$.

   You must decide on a schedule: the order in which to run the jobs. Having fixed an order, each job j has a *completion time* under this order: this is the total amount of time that elapses (from the beginning of the schedule) before it is done being processed.

   For example, suppose you have a set of three jobs $\{1, 2, 3\}$ with

   $$t_1 = 3, \quad t_2 = 1, \quad t_3 = 5,$$

and you run them in this order. Then the completion time of job 1 will be 3, the completion of job 2 will be $3 + 1 = 4$, and the completion time of job 3 will be $3 + 1 + 5 = 9$.

On the other hand, if you run the jobs in the reverse of the order in which they're listed (i.e. 3, 2, 1), then the completion time of job 3 will be 5, the completion of job 2 will be $5 + 1 = 6$, and the completion time of job 1 will be $5 + 1 + 3 = 9$.

(a) Give an algorithm that takes the n processing times $t_1, \ldots, t_n$, and orders the jobs so that the *sum* of the completion times of all jobs is as small as possible. (Such an order will be called *optimal.*)

The running time of your algorithm should be polynomial in n. You should give a complete proof of correctness of your algorithm, and also briefly analyze the running time. As above, you can assume that no two jobs have the same processing time.

(b) Prove that if no two jobs have the same processing time, then the optimal order is *unique.* In other words, for any order other than the one produced by your algorithm in (a), the sum of the completion times of all jobs is not as small as possible.

You may find it helpful to refer to parts of your analysis from (a).

3. Timing circuits are a crucial component of VLSI chips; here's a simple model of such a timing circuit. Consider a complete binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

   The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

   Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem: we want the leaves to be completely synchronized, and all receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain of the edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew.* Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

   Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew, and the total edge length is as small as possible.

   Example. Consider the tree in accompanying figure, with letters naming the nodes and numbers indicating the edge lengths.

   The unique optimal solution for this instance would be to take the three length-1 edges, and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

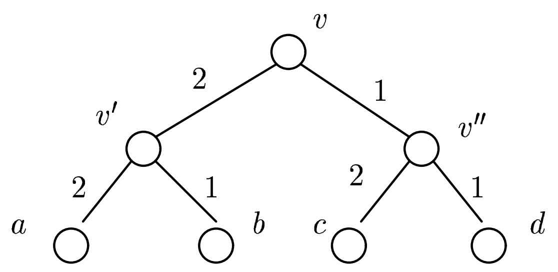

Figure 1: An instance of the zero-skew problem.

4. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

   Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

   The goal in this question is to solve the following algorithmic problem:

   > (*) *Find an independent set in a path G whose total weight (the sum of the weights of the nodes in the independent set) is as large as possible.*

   (a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm:
  Start with $S$ equal to the empty set.
  While some node remains in $G$
    Pick a node $v_i$ of maximum weight.
    Add $v_i$ to $S$.
    Delete $v_i$ and its neighbors from $G$.
  end while
  Return $S$
```

   (b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let $S_1$ be the set of all $v_i$ where $i$ is an odd number.
Let $S_2$ be the set of all $v_i$ where $i$ is an even number.
/* Note that $S_1$ and $S_2$ are both independent sets.  */
Determine which of $S_1$ or $S_2$ has greater total weight,
  and return this one.
```

(c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

5. **Extra Credit:** Give feedback on chapters 3 and 5 of book. Send this by email to Anna and Gideon.